

# Project: research report on MPT

## First. Merkle Patricia Trie 详解

MPT 是默克尔树和帕特里夏树的结合缩写，是一种经过改良的、融合了默克尔树和前缀树两种树结构优点的数据结构，以太坊使用 mpt 存储所有账户状态，以及每个区块中的交易和收据数据。是一种典型的用空间换时间的数据结构。

### 主要功能

- 存储任意长度的 key-value 键值对数据
- 快速计算所维护数据集哈希标识
- 快速状态回滚
- 默克尔证明的证明方法，进行轻节点的扩展，实现简单支付验证

树结构的增删改查操作通常都是递归

### Insert Key Value

### 参数列表

- 当前节点 n
- 节点与待插入的 key 的共同前缀 prefix
- 待插入的 key
- 待插入的 value

### 返回值列表

- 是否是脏数据
- 新节点引用
- 错误消息

### 操作步骤

- 如果待插入的 key 的长度为 0，那么意味着在当前节点上更新待插入的 value;

- 判断当前节点 n 的类型
  - shortNode (压缩节点)
    - 计算当前节点 n 的 key 与待插入的 key 的相同前缀下标+1 (返回的是不一致的第一个下标)
    - 如果相同前缀下标与当前节点 n 的 key 长度一致, 也就意味着待插入的 key 与当前节点 n 的 key 完全匹配, 就是更新当前节点 n 的 value。递归调用当前方法, 参数为当前节点 n 的 value, prefix+之前计算的相同前缀, 待插入的 key 剩下的部分, 以及 value。完成后将返回的节点作为当前节点的 value 返回
    - 如果不一致, 也就意味着有了分支。新建 fullNode, 分别对 当前节点 n 的 key 和 待插入的 key 的不一致下标开始递归调用插入后续 key 及 value, 返回值为 fullNode 的两个分支。节点 n 插入的是 n 的 value, 另一分支为待插入的 value。递归完成后, 当前调用返回 shortNode, key 为相同前缀, value 为新建的 fullNode。还有一种情况, 如果最开始节点 key 就不匹配, 直接就返回 fullNode, 因为没有共同前缀 key
  - fullNode (分支节点)
    - 因为是分支节点, 那么每个 child 的 key 只有一位, 那么只要将 value 插入跟待插入的 key 的第一位相同的 child 位置就可以了。
  - hashNode
    - 哈希节点先去数据库中 load 相关节点的数据, 之后再递归调用
  - 其他
    - 直接将 key, value 包装成 shortNode 返回

## Delete Key

## 参数列表

- 当前节点 n
- 节点与待删除的 key 的共同 prefix
- 待删除的 key

## 返回值列表

- 是否脏数据
- 新节点引用
- 错误信息

## 操作步骤

判断当前节点 n 的类型:

- shortNode
  - 寻找节点 n 的 key 与待删除的 key 的共同前缀下标+1
  - 如果下标小于节点 n 的 key 的长度，表示 key 没匹配上，直接返回
  - 如果下标等于节点 n 的 key 的长度，表示该节点 n 正是需要被删除的节点，删除操作就是返回的新节点引用为 nil，当前节点 n 在内存中就成了野节点，被 mpt 树排除在外了
  - 剩下的情况，表示待删除的 key 存在当前节点 n 的子树，递归调用删除方法，返回后，这里再判断一次返回的新节点 child 的类型：
    - shortNode：父节点 n 与子节点 child 都是一种节点类型，直接合并 key，value 为 child 的 value。
    - 其他：也是返回 shortNode，其 key 为当前节点 n 的 key，value 是递归调用返回的 child
- fullNode
  - 和插入类似，将待删除的 key 的首位与该节点对应的 child 带入递归调用，返回后更新对应 child 的 value
  - 删除之后需要检查分叉节点的所有 child 是否有多于 2 个非 nil 的 child，如果少于 2 个，就可以合并 child
- hashNode：先从数据库 load 数据，在递归调用
- valueNode：直接删除当前节点

## Update Key Value

更新其实是 Insert 与 Delete 的整合

## Get Key Value

## 参数列表

- 当前节点 n
- 待查找的 key
- 已经过滤的子 key 在 key 中的 pos

## 返回值列表

- 查找的 value
- 新节点的引用
- 是否从数据库中 load 数据
- 错误信息

## 操作步骤

判断当前节点 n 的类型：

- valueNode: 直接返回节点 n 的 value
- shortNode
  - 如果剩余的 key (查找的 key 的长度-pos) 的长度小于节点 n 的 key 的长度或者两个 key 的前缀不匹配, 表示在树种没有找到对应的 key, 直接返回
  - 到这里了表示待查找的 key 与该节点 n 的 key 是匹配的, 那么只需要将节点 n 的 value 和剩余的待查找的子 key 带入递归
- fullNode: 同样套路, 找到对应 key 的 child 递归调用
- hashNode: 先载入数据, 在递归调用

以上就是 MPT 树对应的操作, 下面附上改造后的 mpt benchmark:

```
goos: windows
goarch: amd64
BenchmarkHexToCompact-4      500000000      29.6 ns/op
4 B/op      1 allocs/op
BenchmarkCompactToHex-4      300000000      43.5 ns/op
16 B/op      1 allocs/op
BenchmarkKeybytesToHex-4      300000000      44.6 ns/op
32 B/op      1 allocs/op
BenchmarkHexToKeybytes-4      500000000      28.9 ns/op
4 B/op      1 allocs/op
BenchmarkProve-4              30000      44500 ns/op
22816 B/op      679 allocs/op
BenchmarkVerifyProof-4        50000      38020 ns/op
19564 B/op      595 allocs/op
BenchmarkGet-4                1000000      1361 ns/op
512 B/op      25 allocs/op
BenchmarkGetDB-4              1000000      1264 ns/op
512 B/op      25 allocs/op
BenchmarkUpdateBE-4           1000000      3335 ns/op
2145 B/op      32 allocs/op
BenchmarkUpdateLE-4           1000000      4967 ns/op
2195 B/op      32 allocs/op
PASS
ok      68.338s
```

--以上资料来源: <https://zhuanlan.zhihu.com/p/32924994>

## Second. 理解 Substrate 数据存储的底层实现 Merkle Patricia Trie

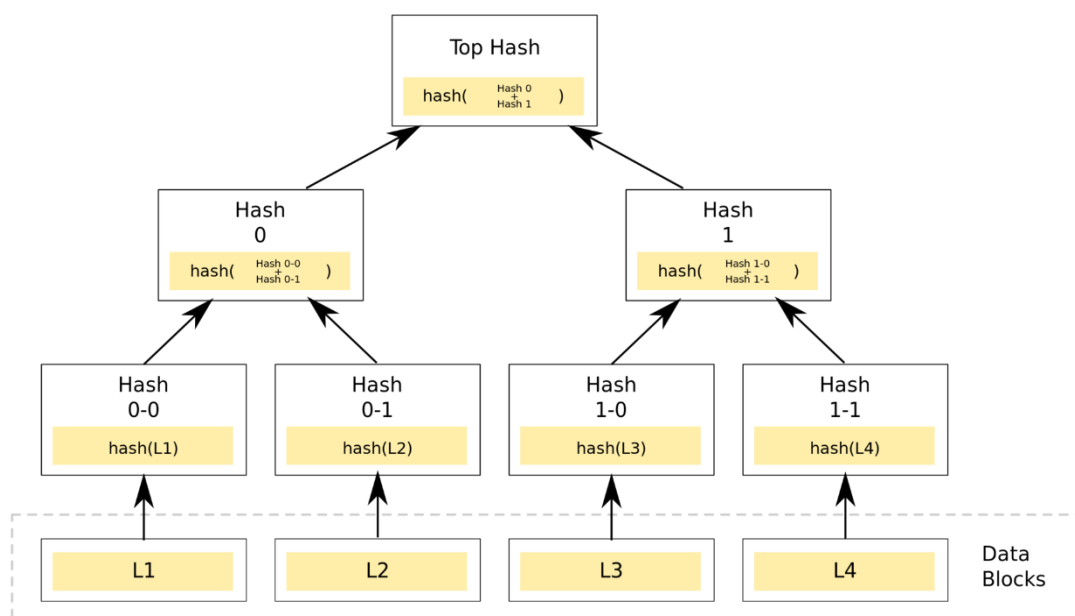
### Merkle Tree 介绍

Merkle Tree 是一种[数据结构](#)，用来验证计算机之间存储和传输数据的一致性，如果不使用这一数据结构，一致性的验证需要消耗大量的存储和网络资源，如比对计算机之间的所有数据；使用 Merkle Tree，只需要比对 merkle root（根节点）就可以达到相同的效果。整个过程，简单的描述如下：

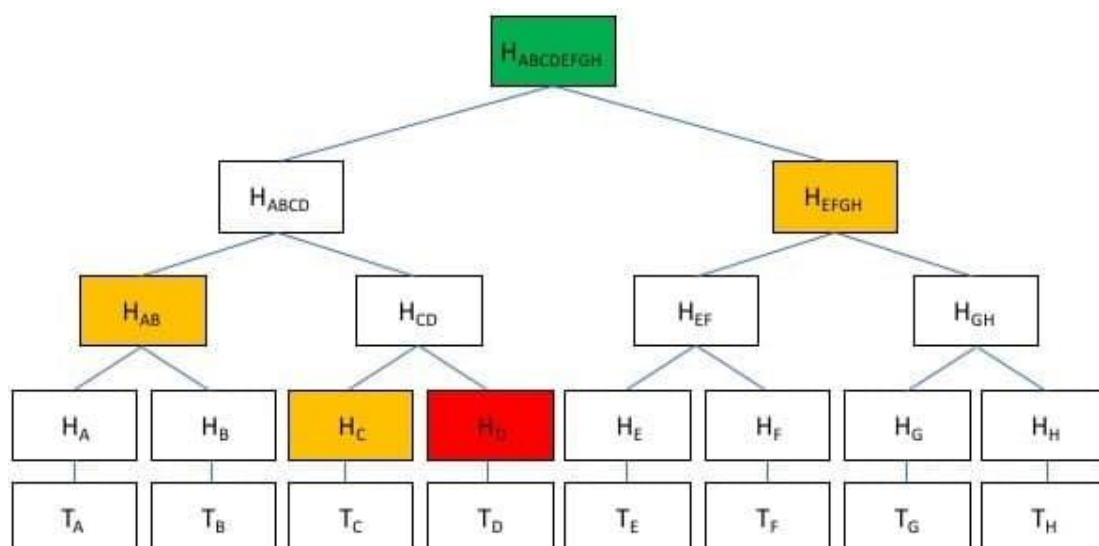
- 将数据通过哈希之后放置在叶子节点之中；
- 将相邻两个数据的哈希值组合在一起，得出一个新的哈希值；
- 依次类推，直到只有一个节点也就是根节点；
- 在验证另外的计算机拥有和本机相同的数据时，只需验证其提供的根节点和自己的根节点一致即可。

Merkle Tree 使用了[加密哈希算法](#)来快速验证数据一致性，常用的加密哈希算法有 SHA-256, SHA-3, Blake2 等，它们可以做到，

- 相同的输入有相同的输出；
- 对任意数据可以实现快速计算；
- 从哈希值无法推断出原信息；
- 不会碰撞（即不同输入对应相同输出）；
- 输入即使只有很小的改变，输出也会有极大不同。



在[区块链](#)应用 Bitcoin 网络中，存储的数据为转移 Bitcoin 的交易，如“Alice 发送给 Bob 5 个比特币”，通过使用 Merkle Tree，除了上面提到的验证各个节点之间的数据一致性，还可以用来快速验证一个交易是否属于某个区块。轻节点只需要下载很少的数据就可以验证交易的有效性，例如下图所示，用户要验证交易 T(D) 在某个区块之中，需要依赖的数据仅仅是 HC，HAB，HEFGH，和 merkle root 即 HABCDEFGH。



## Merkle Patricia Trie 原理

### Trie

[Patricia Trie](#) 也是一种树形的数据结构，也称为 Prefix Tree，Radix Tree，或者简称为 Trie，最早来源于英文单词 **re**trieve，可以发音为 *try*，常用的使用场景包括：

- 搜索引擎的自动补全功能；
- IP 路由等。



Google Search

I'm Feeling Lucky

Trie 的特点是，某节点的 key 是从根节点到该节点的路径，即不同的 key 有相同前缀时，它们共享前缀所对应的路径。这种数据结构，可用于快速查找前缀相同的数据，内存开销较少。如以下数据及对应的 trie 表示为：

**key value**

to 7

tea 3

ted 4

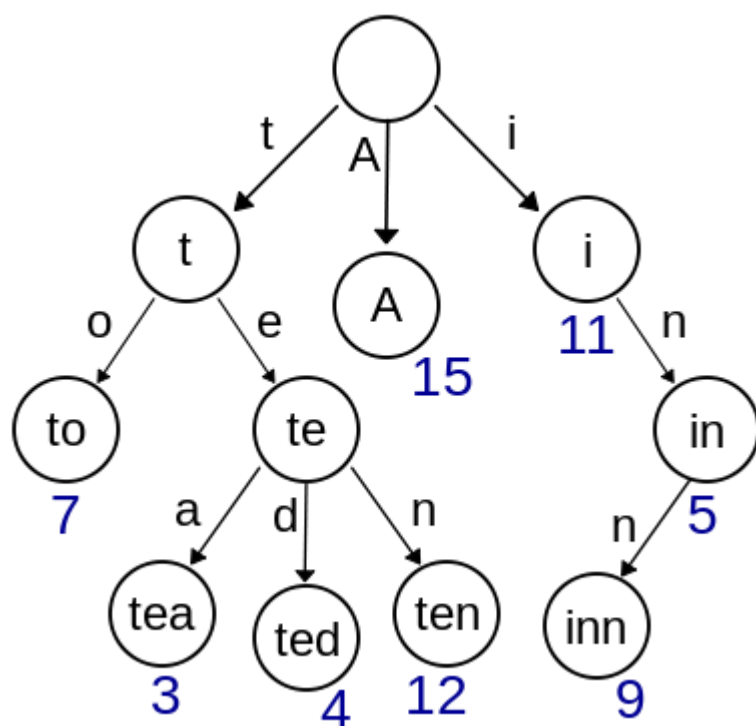
ten 12

A 15

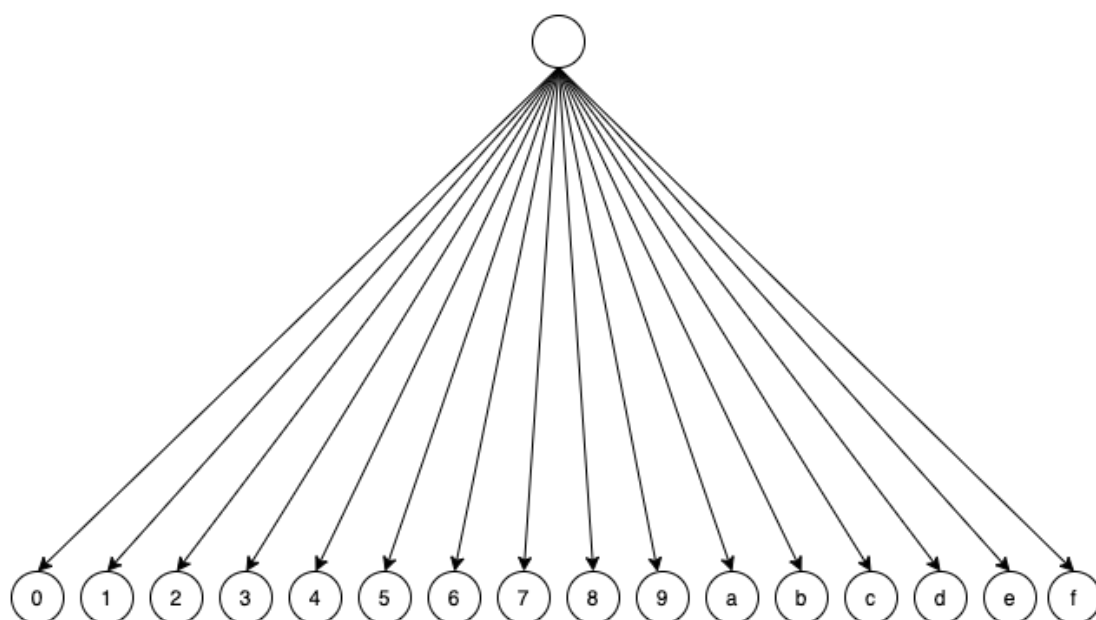
i 11

in 5

inn 9



Substrate 使用 base-16，即每个节点最多有 16 个子节点：





## MPT

Merkle Patricia Trie（下面简称 MPT），在 Trie 的基础上，给每个节点计算了一个哈希值，在 Substrate 中，该值通过对节点内容进行 Blake2 运算取得，用来索引数据库和计算 merkle root。也就是说，MPT 用到了两种 key 的类型。

一种是 Trie 路径所对应的 key，由 runtime 模块的存储单元决定。使用 Substrate 开发的应用链，它所拥有的各个模块的存储单元会通过交易进行修改，成为链上状态（简称为 state）。每个存储单元的状态都是通过键值对以 trie 节点的形式进行索引或者保存的，这里键值对的 value 是原始数据（如数值、布尔）的 SCALE 编码结果，并作为 MPT 节点内容的一部分进行保存；key 是模块、存储单元等的哈希组合，且和存储数据类型紧密相关，如：

- 单值类型（即 Storage Value），它的 key 是 `Twox128(module_prefix) ++ Twox128(storage_prefix);`
- 简单映射类型（即 map），可以表示一系列的键值数据，它的存储位置和 map 中的键相关，即 `Twox128(module_prefix) + Twox128(storage_prefix) + hasher(encode(map_key));`
- 链接映射类型（即 linked\_map），和 map 类似，key 是 `Twox128(module_prefix) + Twox128(storage_prefix) + hasher(encode(map_key));` 它的 head 存储在 `Twox128(module) + Twox128("HeadOf" + storage_prefix);`
- 双键映射类型（即 double\_map），key 是 `twox128(module_prefix) + twox128(storage_prefix) + hasher1(encode(map_key1)) + hasher2(encode(map_key2))。`

计算 key 所用到 [Twox128](#) 是一种非加密的哈希算法，计算速度非常快，但去除了一些严格的要求，如不会碰撞、很小的输入改变导致极大的输出改变等，从而无法保证安全性，适用于输入固定且数量有限的场景中。module\_prefix 通常是模块的实例名称；storage\_prefix 通常是存储单元的名称；原始的 key 通过 SCALE 编码器进行编码，再进行哈希运算，这里的哈希算法是可配置的，如果输入来源不可信如用户输入，则使用 Blake2（也是默认的哈希算法），否则可以使用 Twox。

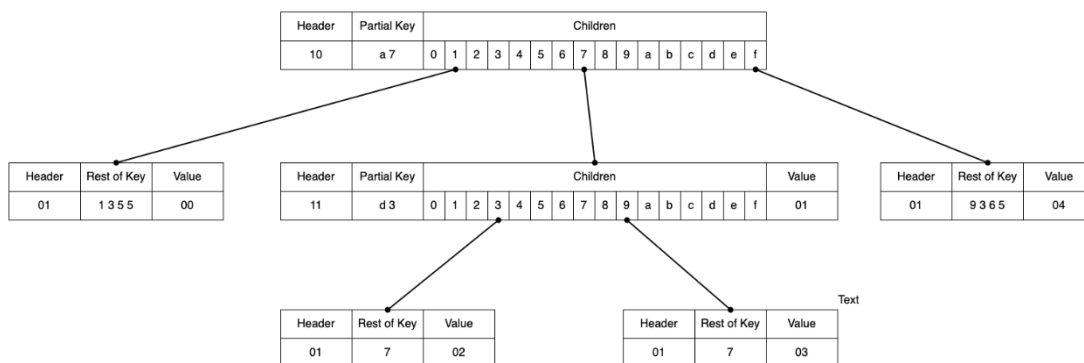
另一种是数据库存储和计算 merkle root 使用的 key，可以通过对节点内容进行哈希运算得到，在键值数据库（即 RocksDB，和 LevelDB 相比，[RocksDB 有更多的性能优化和特性](#)）中索引相应的 trie 节点。

Trie 节点主要有三类，即叶子节点（Leaf）、有值分支节点（BranchWithValue）和无值分支节点（BranchNoValue）；有一个特例，当 trie 本身为空的时候存在唯一的空节点（Empty）。根据类型不同，trie 节点存储内容有稍许不同，通常会包含 header、trie 路径的部分 key、children 节点以及 value。下面举一个具体例子。

*Trie 路径的 key 和 value 如下表所示:*

| Trie Path Key |   |   |   |   |   |   | Value |
|---------------|---|---|---|---|---|---|-------|
| a             | 7 |   |   |   |   |   | N/A   |
| a             | 7 | 1 | 1 | 3 | 5 | 5 | 00    |
| a             | 7 | 7 | d | 3 |   |   | 01    |
| a             | 7 | 7 | d | 3 | 3 | 7 | 02    |
| a             | 7 | 7 | d | 3 | 9 | 7 | 03    |
| a             | 7 | f | 9 | 3 | 6 | 5 | 04    |

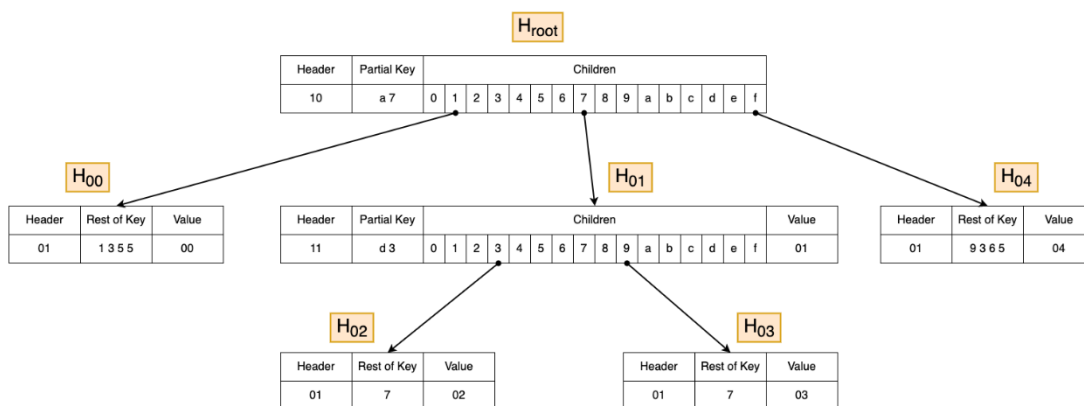
*将上表的数据展示为 trie 的结构之后得到下图:*



图上所示内容的说明如下：

- 这里使用不同的 header 值来表示不同的节点类型，即 01 表示叶子节点，10 表示无值的分支节点，11 表示有值的分支节点。
- 为了提高存储和查找的效率，Substrate 使用的 MPT 和基本的 trie 不同的是，并不是 trie path key 的每一位都对应一个节点，当 key 的连续位之间没有分叉时，节点可以直接使用该连续位。对于叶子节点，连续位可以是 trie path key 的最后几位；对于分支节点，连续位是从当前位开始到出现分叉位结束。
- 之前提到过，Substrate 采用了 base-16 的 trie 结构，即一个分支节点最多可以有 16 个子节点。

接着，我们来添加每个节点的哈希：



首先计算出叶子节点的哈希值，它们被上一级的分支节点所引用，用来在数据库中查找对应的节点；然后计算分支节点的哈希值，直至递归抵达根节点，这里用到的哈希算法是 Blake2。

数据库的物理存储大致如下：

| key   | value             |
|-------|-------------------|
| Hroot | encode(root_node) |
| H00   | encode(node_00)   |
| H01   | encode(node_01)   |
| H02   | encode(node_02)   |
| H03   | encode(node_03)   |
| H04   | encode(node_04)   |

数据库中存储的 key 是上面所计算的节点哈希；存储的 value 是节点内容的特定编码，对于节点中保存的值是对应的 SCALE 编码结果。

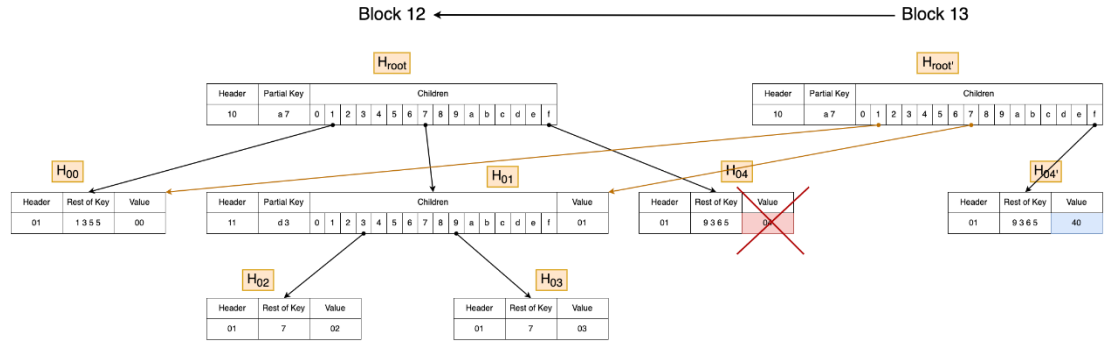
RocksDB 提供了 column 的概念，用来存储互相隔离的数据。例如，HEADER column 存储着所有的区块头；BODY column 存储着所有的区块体，也就是所有的存储单元状态。

# Child Trie

Substrate 还提供 child trie 这样的数据结构，它也是一个 MPT。Substrate 的应用链可以有多个 child trie，每个 child trie 有唯一的标识进行索引，它们可以彼此隔离，提升存储和查询效率。State trie 或者 main trie 的叶子节点可以是 child trie 的根哈希，来保存对应 child trie 的状态。Child trie 的一个典型应用是 Substrate 的 contracts 功能模块，每一个智能合约对应着自己唯一的 child trie。

# 区块裁剪

区块的无限增长往往给去中心的节点造成较大的存储消耗，通常只有少数的节点需要提供过往历史数据，大部分节点在能够确保状态最终性之后，可以将不需要的区块数据删除，从而提升节点的性能。Substrate 也内置了裁剪的功能，示意图如下：



在区块 13 中，只有 node-4 的值从 04 变为 40，其哈希也跟着改变，而其它节点的数据并没有变化，所以新的区块根节点可以复用原有节点，仅仅更新发生变化的节点。假设区块 13 已经具有最终性，那么区块 12 中的 node-4 就可以删掉。Substrate 默认的区块生成算法是 BABE 或者 Aura，而最终性是通过 GRANDPA 来决定的，在网络稳定的情况下，仅保留一定数量的最新区块是可行的。

## 总结

本文介绍了区块链应用必不可少 Merkle Tree，以及 Substrate 采用的 Patricia Merkle Trie 的不同之处。需要说明的是，Substrate 还提供了强大的 cache 功能，来提升读写效率。

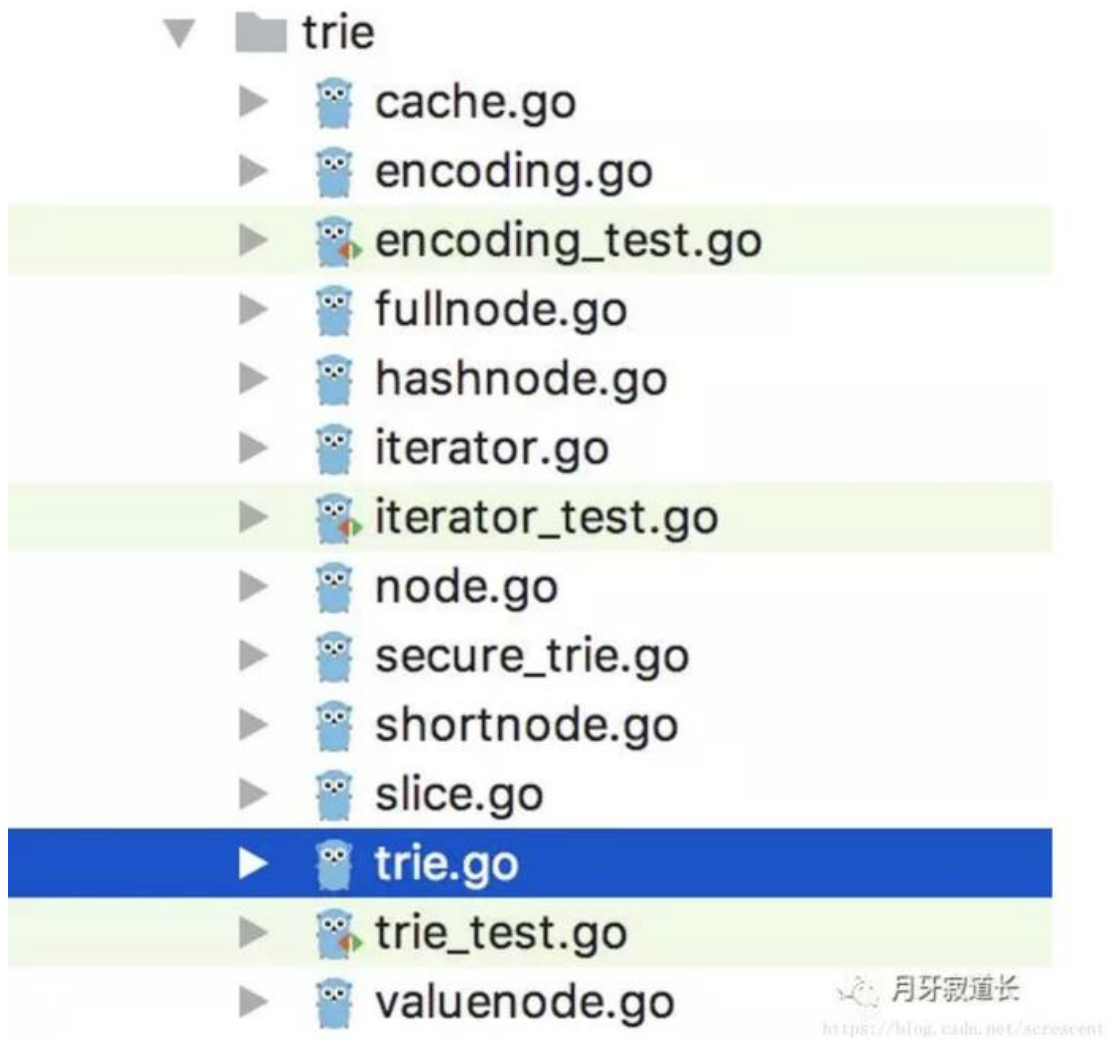
--以上资料来源：

<https://blog.csdn.net/shangsongwww/article/details/119272573>

## Third. 以太坊源码分析 ---go-ethereum 之 MPT(Merkle-Patricia Trie)

那么 MPT 呢，是以太坊中，自定义的数据结构。综合了 Merkle Tree 与 Patricia Trie 两个的特点。

那么看源码先吧。注：代码为 [github.com/ethereum/go-ethereum/trie](https://github.com/ethereum/go-ethereum/trie)，版本为 1.0.0



这里为 trie 的整个 mpt 代码模块。顺便说下，以太坊的源码结构，模块化非常好。


首先介绍 node 概念

[github.com/ethereum/go-ethereum/trie/node.go](https://github.com/ethereum/go-ethereum/trie/node.go)

```

22
23 type Node interface {
24     Value() Node
25     Copy(*Trie) Node // All nodes, for now, return them self
26     Dirty() bool
27     fstring(string) string
28     Hash() interface{}
29     RlpData() interface{}
30     setDirty(dirty bool)
31 }

```

 月牙寂道长  
<https://blog.cadu.net/screencn1>

node interface 定义

那么分了几类 node


## 1、fullnode

[github.com/ethereum/go-ethereum/trie/fullnode.go](https://github.com/ethereum/go-ethereum/trie/fullnode.go)

```

18
19 type FullNode struct {
20     trie *Trie
21     nodes [17]Node
22     dirty bool
23 }

```

 月牙寂道长  
<https://blog.cadu.net/screencn1>

- trie 指向的是 root
- 一个容量为 17 的子节点。其中前 16 个代表的是 0-9a-f 的子节点索引，这个和 trie 树是一样的。第 17 个则用于保存该 fullnode 的数据。
- dirty 用于标识 trie 树是否发生变化。

fullnode 用于无法合并的情况下的节点，那么会有完整的子节点索引（0-9a-f）。

## 2、shortnode

[github.com/ethereum/go-ethereum/trie/shortnode.go](https://github.com/ethereum/go-ethereum/trie/shortnode.go)

```
20
21 ①↑ type ShortNode struct {
22     trie *Trie
23     key  []byte
24     value Node
25     dirty bool
26 }
27
```

月牙寂道长  
<https://blog.cade.net/arcoscm/>

- trie 指向的是 root
- key 是一个数组，这里的 key 则是和 Patricia Trie 中一样，属于合并节点，用于优化存储空间和查询效率。合并原则是当父节点只有一个子节点的时候，将其合并。
- value 指向的是一个节点。
- dirty 用于标识 trie 树是否有发生变化。

shortnode 用于优化合并节点，所以其 key 是合并结果的 key。

fullnode 和 shortnode 为结构性节点，搞清楚这两个节点的区别，就基本上搞定了 mpt 的数据结构。

那么除了结构性节点，还需要数据节点。

## 1、valuenode


[github.com/ethereum/go-ethereum/trie/valuenode.go](https://github.com/ethereum/go-ethereum/trie/valuenode.go)



```

20
21 type ValueNode struct {
22     trie *Trie
23     data []byte
24     dirty bool
25 }
26

```

 月牙寂道长  
<https://blog.csdn.net/acrescent>

- trie 指向 root
- data 为具体的数据
- dirty 用于标识 trie 树是否有发生变化。

valuenode 很简单，只是用于存储 value，无其他。

## 2、hashnode

[github.com/ethereum/go-ethereum/trie/hashnode.go](https://github.com/ethereum/go-ethereum/trie/hashnode.go)

```

20
21 type HashNode struct {
22     key []byte
23     trie *Trie
24     dirty bool
25 }
26

```

 月牙寂道长  
<https://blog.csdn.net/acrescent>

- key 为 hashnode 保存的要索引的 key。一般为这里的 key 对应的数据保存在数据库中，还未加载。
- trie 指向 root
- dirty 用于标识 trie 树是否有发生变化。

当 trie 加载的时候，并不需要全部加载，未加载的部分，可以将其用 hashnode 表达。

valuenode 和 hashnode 为数据节点，只用于存储 value。并不参与结构性。

那么从功能上区分结构性节点和数据节点，就是此数据结构的重心。

结构性节点：

fullnode

shortnode

数据性节点

valuenode

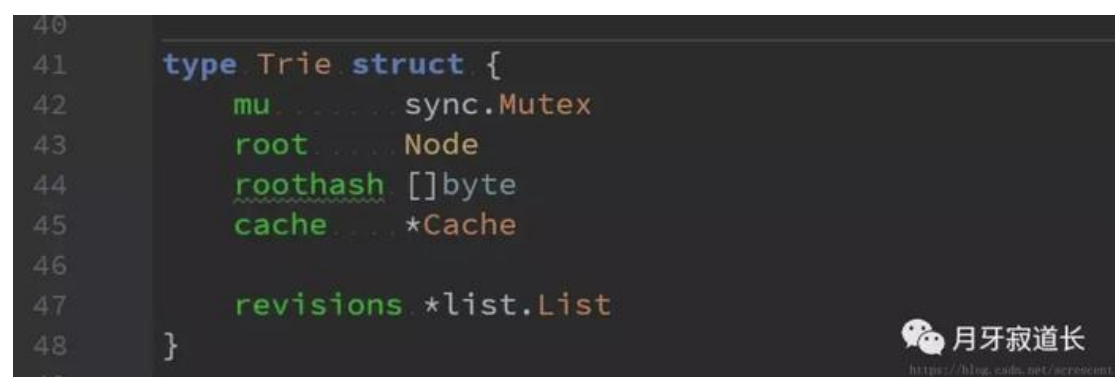
hashnode

了解了结构后，那么就需要看流程了。

## Trie 结构

[github.com/ethereum/go-ethereum/trie/trie.go](https://github.com/ethereum/go-ethereum/blob/master/trie/trie.go)

```
40
41 type Trie struct {
42     mu      sync.Mutex
43     root    Node
44     roothash []byte
45     cache   *Cache
46
47     revisions *list.List
48 }
```



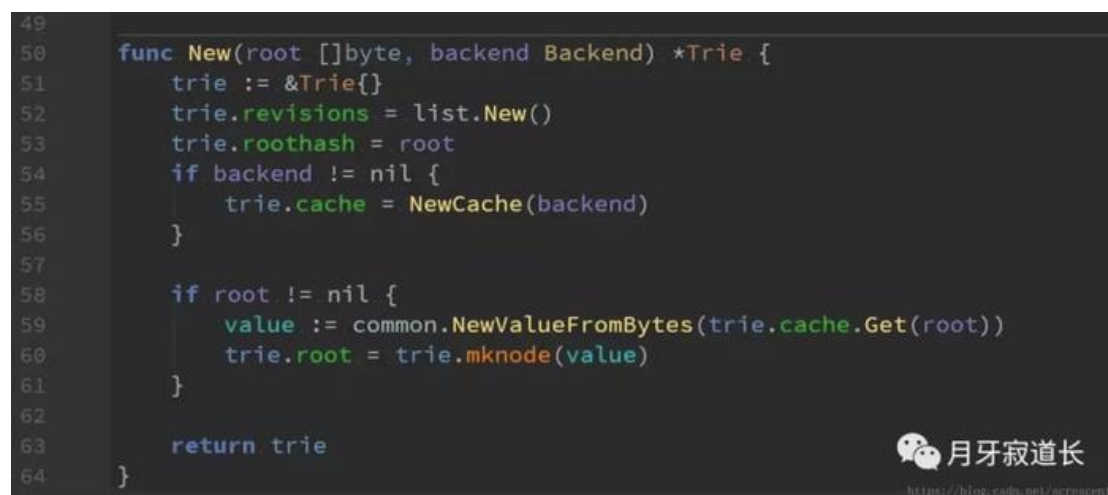
- mu 为锁
- root 为根 node
- roothash 保存 root 对应的 hash
- cache 用于保存 node 的 cache

- revisions 为保存 roothash 的版本变化

## New

创建一个新的 Trie。这里的流程是这样的

```
49
50 func New(root []byte, backend Backend) *Trie {
51     trie := &Trie{}
52     trie.revisions = list.New()
53     trie.roothash = root
54     if backend != nil {
55         trie.cache = NewCache(backend)
56     }
57
58     if root != nil {
59         value := common.NewValueFromBytes(trie.cache.Get(root))
60         trie.root = trie.mknnode(value)
61     }
62
63     return trie
64 }
```



51: 创建一个结构体

52: 初始化 revisions, 用于保存 roothash, 记录 roothash 变化

53: 将 root 赋值给 roothash (说明参数 root 是一个 hash)

54-56: 创建一个 cache

58-61: 当 root 为 nil 的时候, 说明创建一个空的 Trie。

当 root 不为 nil 的时候, 说明为加载一个已经存在的 Trie。

从最初的流程来看, 应该是创建一个空的为最开始的。(这个很关键, 是我们跟踪流程的起始)

## 插入分析

## Update

```
130 func (self *Trie) Update(key, value []byte) Node {
131     self.mu.Lock()
132     defer self.mu.Unlock()
133
134     k := CompactHexDecode(string(key))
135
136     if len(value) != 0 {
137         node := NewValueNode(self, value)
138         node.dirty = true
139         self.root = self.insert(self.root, k, node)
140     } else {
141         self.root = self.delete(self.root, k)
142     }
143
144     return self.root
145 }
```

月牙寂道长  
<https://blog.csdn.net/screaom>

131-132: 锁操作

134: key 的转换

136-139: 当 value 不为空的时候, 说明是要刷新节点或者插入一个新的节点。

创建一个 valuenode, 并 dirty 为 true。

调用 insert。

141: 当 value 为空的时候, 说明是要将节点删除, 调用了 delete。

## insert

```
172
173 func (self *Trie) insert(node Node, key []byte, value Node) Node {
174     if len(key) == 0 {
175         return value
176     }
177
178     if node == nil {
179         node := NewShortNode(self, key, value)
180         node.dirty = true
181         return node
182     }
183 }
```

月牙寂道长  
<https://blog.csdn.net/screaom>

178-181: 当 node 为 nil 的时候。

当我们的 trie 为空的时候，root 就是为 nil。那么第一次插入的时候，走的就是这个流程。

创建了 shortnode。并将其返回了。

在 Update 函数中，trie 的 root 会被赋值为这个 shortnode。

那么第一次的插入就完成了。

```
183
184     switch node := node.(type) {
185     case *ShortNode:
186         k := node.Key()
187         cnode := node.Value()
188         if bytes.Equal(k, key) {
189             node := NewShortNode(self, key, value)
190             node.dirty = true
191             return node
192         }
193     }
194
195     var n Node
196     matchlength := MatchingNibbleLength(key, k)
197     if matchlength == len(k) {
198         n = self.insert(cnode, key[matchlength:], value)
199     } else {
200         pnode := self.insert(node: nil, k[matchlength+1:], cnode)
201         nnode := self.insert(node: nil, key[matchlength+1:], value)
202         fulln := NewFullNode(self)
203         fulln.dirty = true
204         fulln.set(k[matchlength], pnode)
205         fulln.set(key[matchlength], nnode)
206         n = fulln
207     }
208     if matchlength == 0 {
209         return n
210     }
211
212     snode := NewShortNode(self, key[:matchlength], n)
213     snode.dirty = true
214     return snode
215
```

月牙寂道长  
<https://blog.csdn.net/serresent>

再继续插入的话，执行流程就走到这里了。第一次插入的 root 肯定是一个 shortnode。

186-193: 对 key 进行判断, 如果 key 与要插入的 node 的 key 是一致的话, 只需要构建一个新的 shortnode, 进行替换即可。

下面的是很重要的插入流程。

196: 变量 k 为插入节点 node 的 node\_key (为了区分变量), 变量 key 为要被插入节点的 node\_key。计算其 k 和 key 的重合度。如重合度相同则直接插入。

200-206:

说明 k 和 key 的长度不一样, 那么需要将 shortnode 拆分裂变为两个。

pnode, nnode (这里面还有递归插入后续的动作)

而当前的 node 则替换成 fullnode, 并将两个子节点 set 到这个 fullnode 下。

208-210: 这里判断匹配长度是不是为 0, 是则, 此次递归就完成了。

212: 如果递归还没未完成, 说明还有遗留的部分 key 是无法匹配的, 那么就创建一个 shortnode 来包容剩下的。

```
116     case *FullNode:
117         cpy := node.Copy(self).(*FullNode)
118         cpy.set(key[0], self.insert(node.branch(key[0]), key[1:], value))
119         cpy.dirty = true
120
121         return cpy
122
123     default:
124         panic(fmt.Sprintf("invalid node: %v", node, node))
125 }
126 }
```

月牙寂道长  
<https://blog.csdn.net/serenacat>

当插入节点处为 fullnode 的时候, 插入就比较简单。


218: 直接将其插入到 fullnode 中。

但也继续递归插入了剩下的部分。

## 查询分析

Get、GetString

```
147 func (self *Trie) GetString(key_string) []byte { return self.Get([]byte(key)) }
148 func (self *Trie) Get(key []byte) []byte {
149     self.mu.Lock()
150     defer self.mu.Unlock()
151
152     k := CompactHexDecode(string(key))
153
154     n := self.get(self.root, k)
155     if n != nil {
156         return n.(*ValueNode).Val()
157     }
158
159     return nil
160 }
```



月牙寂道长


<https://blog.csdn.net/serosum>

154: 主要过程在这里调用了 get，从 root 节点开始查找

```

228 func (self *Trie) get(node Node, key []byte) Node {
229     if len(key) == 0 {
230         return node
231     }
232
233     if node == nil {
234         return nil
235     }
236
237     switch node := node.(type) {
238     case *ShortNode:
239         k := node.Key()
240         cnode := node.Value()
241
242         if len(key) >= len(k) && bytes.Equal(k, key[:len(k)]) {
243             return self.get(cnode, key[len(k):])
244         }
245
246         return nil
247     case *FullNode:
248         return self.get(node.branch(key[0]), key[1:])
249     default:
250         panic(fmt.Sprintf("format: invalid node: %v", node, node))
251     }
252 }

```

 月牙寂道长  
<https://blog.csdn.net/acrescent>

229: 为退出的条件, 当 key 被递归查询到为 0 的时候, 则返回当前 node。

237 开始, 根据类型进行不同的查找

239-246: shortnode, 不断的进行 key 匹配查找, 递归查询到最后的子节点。

248: 递归查下下一个 fullnode 节点。

```

82
83 func (self *FullNode) branch(i byte) Node {
84     if self.nodes[int(i)] != nil {
85         self.nodes[int(i)] = self.trie.trans(self.nodes[int(i)])
86
87         return self.nodes[int(i)]
88     }
89     return nil
90 }

```

 月牙寂道长  
<https://blog.csdn.net/acrescent>

这个为 fullnode 子节点索引查找。



那么最重要的 mpt 的数据结构的插入和查询就完成了。如果能够把这些弄明白的话，那么就对 mpt 有了个很深刻的理解了。

本文的重点是对 mpt 有个简单的讲解。

在 Trie 源码中，还有 cache、encoding、iterator 代码，就不一一解释。

cache 是一个简答的缓存，里面有封装了有 db、map 等。

encoding 是一个编码的转换

iterator 是一个迭代器

--资料来源: <https://www.likecs.com/show-204048436.html>

## Fourth. LMPTs: Eliminating Storage Bottlenecks for Processing Blockchain Transactions

JA Choi、SM Beillahi、P. Li、A. Veneris 和 F. Long, “LMPT: 消除处理区块链交易的存储瓶颈”, 2022 年 IEEE 区块链和加密货币国际会议 (ICBC) , 2022 年 , 第 1-9 页 , doi : 10.1109/ICBC54727.2022.9805484. Abstract: We present the Layered

Merkle Patricia Trie (LMPT), a performant storage data structure for processing transactions in high-throughput systems when compared to traditional Merkle Patricia Tries used in Ethereum clients. LMPTs keep smaller intermediary tries in memory to alleviate read and write amplification from high-latency disk storage. As an additional feat, they also allow for the I/O and transaction verifier threads to be scheduled in parallel and independently. LMPTs can ultimately reduce significant I/O traffic that happens on the critical path of transaction processing. Empirical results presented here confirm that LMPTs can process up to  $\times 6$  more transactions per second on real-life workloads when compared to existing Ethereum clients.

URL: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9805484&isnumber=9805482>